Traitement du signal et Apprentissage profond ; applications industrielles

Thomas COURTAT
thomas.courtat@thalesgroup.com

Master MVA 2023

Plan du cours

1. Apprentissage profond
   - Rappels d'apprentissage machine et d'optimisation
   - Réseaux de neurones et Graphes de calcul
   - Apprentissage de Réseaux de neurones
   - Modèles et couches de réseaux de neurones - niveau 1
   - Trucs et astuces
   - Application : reconnaissance vocale
   - Mise en oeuvre avec PyTorch

Apprentissage profond

Rappels d'apprentissage machine et d'optimisation

Algorithme paramétrique

Un *algorithme* est une procédure qui à partir de données x dans une espace $\mathcal{E}_X$ produit des résultats y dans un autre espace $\mathcal{E}_Y$:

$$A : \mathcal{E}_X \to \mathcal{E}_Y$$

Algorithme paramétrique

Un *algorithme* est une procédure qui à partir de données x dans une espace $\mathcal{E}_X$ produit des résultats y dans un autre espace $\mathcal{E}_Y$:

$$A : \mathcal{E}_X \to \mathcal{E}_Y$$

Notamment, deux types d'algorithmes:

Algorithme paramétrique

Un *algorithme* est une procédure qui à partir de données x dans une espace $\mathcal{E}_X$ produit des résultats y dans un autre espace $\mathcal{E}_Y$:

$$A : \mathcal{E}_X \rightarrow \mathcal{E}_Y$$

Notamment, deux types d'algorithmes:
- Classification sur C classes $\Rightarrow$ $\mathcal{E}_Y$ est un ensemble discret de taille C

Algorithme paramétrique

Un *algorithme* est une procédure qui à partir de données x dans une espace $\mathcal{E}_X$ produit des résultats y dans un autre espace $\mathcal{E}_Y$:

$$A : \mathcal{E}_X \rightarrow \mathcal{E}_Y$$

Notamment, deux types d'algorithmes:
- Classification sur C classes $\Rightarrow$ $\mathcal{E}_Y$ est un ensemble discret de taille C
    - Classification souple: $\mathcal{E}_Y$ est l'ensemble des vecteurs de dimension C, $y \in \mathcal{E}_Y \Rightarrow \forall i, y_i \geq 0, \sum_i y_i = 1$

Algorithme paramétrique

Un *algorithme* est une procédure qui à partir de données x dans une espace $\mathcal{E}_X$ produit des résultats y dans un autre espace $\mathcal{E}_Y$:

$$A : \mathcal{E}_X \to \mathcal{E}_Y$$

Notamment, deux types d'algorithmes:
- Classification sur C classes $\Rightarrow$ $\mathcal{E}_Y$ est un ensemble discret de taille C
    - Classification souple: $\mathcal{E}_Y$ est l'ensemble des vecteurs de dimension C, $y \in \mathcal{E}_Y \Rightarrow \forall i, y_i \geq 0, \sum_i y_i = 1$
- Régression $\Rightarrow$ $\mathcal{E}_Y$ est un $\mathbb{R}$ espace vectoriel

Algorithme paramétrique

Un *algorithme* est une procédure qui à partir de données x dans une espace $\mathcal{E}_X$ produit des résultats y dans un autre espace $\mathcal{E}_Y$:

$$A : \mathcal{E}_X \to \mathcal{E}_Y$$

Notamment, deux types d'algorithmes:
- Classification sur C classes $\Rightarrow$ $\mathcal{E}_Y$ est un ensemble discret de taille C
    - Classification souple: $\mathcal{E}_Y$ est l'ensemble des vecteurs de dimension C, $y \in \mathcal{E}_Y \Rightarrow \forall i, y_i \geq 0, \sum_i y_i = 1$
- Régression $\Rightarrow$ $\mathcal{E}_Y$ est un $\mathbb{R}$ espace vectoriel

Un *Algorithme paramétrique* est un algorithme A_θ dont la procédure dépend de paramètres regroupés dans un vecteur θ.

Apprentissage machine supervisé

L'apprentissage supervisé est un domaine qui permet:

Apprentissage machine supervisé

L'apprentissage supervisé est un domaine qui permet:
- de choisir une certaine classe d'algorithmes A_θ

Apprentissage machine supervisé

L'apprentissage supervisé est un domaine qui permet:
- de choisir une certaine classe d'algorithmes A_θ
- de régler les paramètres de θ rationnellement sur un ensemble de *données d'apprentissage annotées* $\{(x_i, y_i)\}$ dans une *phase d'apprentissage*

Apprentissage machine supervisé

L'apprentissage supervisé est un domaine qui permet:
- de choisir une certaine classe d'algorithmes A_θ
- de régler les paramètres de θ rationnellement sur un ensemble de *données d'apprentissage annotées* $\{(x_i, y_i)\}$ dans une *phase d'apprentissage*
- une fois θ réglé et fixé à θ_*, de *prédire* les valeurs y pour de nouvelles données x, dans une *phase d'inférence*

Apprentissage machine supervisé

L'apprentissage supervisé est un domaine qui permet:
- de choisir une certaine classe d'algorithmes A_θ
- de régler les paramètres de θ rationnellement sur un ensemble de *données d'apprentissage annotées* $\{(x_i, y_i)\}$ dans une *phase d'apprentissage*
- une fois θ réglé et fixé à θ_*, de *prédire* les valeurs y pour de nouvelles données x, dans une *phase d'inférence*

La phase d'apprentissage en général par maximisation d'une fonction d'adéquation entre les sorties de l'algorithme évalué sur les données d'apprentissage et les annotations:

$$\theta_* \in \text{Argmin}_\theta \, \mathcal{L}((A_\theta(x_i))_i, (y_i)_i)$$

$\mathcal{L}$ fonction de perte, minimale si adéquation maximale.

Régression logistique

Classifieur binaire probabiliste:
- Entrée = vecteur de dimension M,
- Sortie = vecteur stochastique de dimension 2: $A_\theta(x) = \begin{pmatrix} A_\theta(x)_0 \\ A_\theta(x)_1 \end{pmatrix}$, $A_\theta(x)_i$, $i = 0, 1$

est l'estimation de $P(Y(X) = i|X)$.

Régression logistique

Classifieur binaire probabiliste:
- Entrée = vecteur de dimension M,
- Sortie = vecteur stochastique de dimension 2: $A_\theta(x) = \begin{pmatrix} A_\theta(x)_0 \\ A_\theta(x)_1 \end{pmatrix}$, $A_\theta(x)_i$, $i = 0, 1$

est l'estimation de $P(Y(X) = i|X)$.

Hypothèse: $\forall i$, $\log P(X = x|Y(X) = i) = \sum_j \theta_j^{(i)} x_j + \text{Cste}$

$\Rightarrow P(Y(X) = 0|X = x) = \frac{\exp \sum_j \delta_j x_j}{1+\exp \sum_j \delta_j x_j}$, $\delta_j = (\theta_j^{(0)} - \theta_j^{(1)})$

Régression logistique

Classifieur binaire probabiliste:
- Entrée = vecteur de dimension M,
- Sortie = vecteur stochastique de dimension 2: $A_\theta(x) = \begin{pmatrix} A_\theta(x)_0 \\ A_\theta(x)_1 \end{pmatrix}$, $A_\theta(x)_i$, $i = 0, 1$

est l'estimation de $P(Y(X) = i|X)$.

Hypothèse: $\forall i$, $\log P(X = x|Y(X) = i) = \sum_j \theta_j^{(i)} x_j + \text{Cste}$

$\Rightarrow P(Y(X) = 0|X = x) = \frac{\exp \sum_j \delta_j x_j}{1+\exp \sum_j \delta_j x_j}$, $\delta_j = (\theta_j^{(0)} - \theta_j^{(1)})$

Fonction de perte: - log vraisemblance des données

$$\mathcal{L}((A_\theta(x_i))_i, (y_i)_i) = - \sum_{i \in \text{toutes les données}} \log A_\theta(x_i)_{y_i}$$

Régression logistique

Classifieur binaire probabiliste:
- Entrée = vecteur de dimension M,
- Sortie = vecteur stochastique de dimension 2: $A_\theta(x) = \begin{pmatrix} A_\theta(x)_0 \\ A_\theta(x)_1 \end{pmatrix}$, $A_\theta(x)_i$, $i = 0, 1$

est l'estimation de $P(Y(X) = i|X)$.

Hypothèse: $\forall i$, $\log P(X = x|Y(X) = i) = \sum_j \theta_j^{(i)} x_j + \text{Cste}$

$\Rightarrow P(Y(X) = 0|X = x) = \frac{\exp \sum_j \delta_j x_j}{1+\exp \sum_j \delta_j x_j}$, $\delta_j = (\theta_j^{(0)} - \theta_j^{(1)})$

Fonction de perte: - log vraisemblance des données

$$\mathcal{L}((A_\theta(x_i))_i, (y_i)_i) = - \sum_{i \in \text{toutes les données}} \log A_\theta(x_i)_{y_i}$$

Minimisation de $\mathcal{L}$ par descente de gradient .

Descente de gradient

$f(\theta)$ à optimiser, on construit la suite (θ_i) avec

$$\theta_{i+1} = \theta_i - \mu \frac{\partial f}{\partial \theta}(\theta_i)$$

.

$\mu > 0$ est un paramètre à fixer.

$\Rightarrow$ Si μ bien choisi, (θ_i) converge vers un minimum local de f.

Descente de gradient

$f(\theta)$ à optimiser, on construit la suite (θ_i) avec

$$\theta_{i+1} = \theta_i - \mu \frac{\partial f}{\partial \theta}(\theta_i)$$

.
$\mu > 0$ est un paramètre à fixer.
$\Rightarrow$ Si μ bien choisi, (θ_i) converge vers un minimum local de f.

Pour la régression logistique, on prend $f(\theta) = -\sum_{i \in \text{toutes les données}} \log A_\theta(x_i)_{y_i}$

Descente de gradient

On peut exploiter les dérivées d'ordre 2:

$f(\theta + \delta\theta) \simeq f(\theta) + \quad {}^t\frac{\partial f}{\partial \theta}(\theta_i) \cdot \delta\theta + \frac{1}{2}{}^t\delta\theta H \delta\theta$

Si θ de dimension n H matrice de taille $n \times n$ (Hessienne), $H_{kl} = \frac{\partial^2 f}{\partial \theta_k \partial \theta_l}$

On prend

$$\theta_{i+1} = \theta_i - H^{-1}\frac{\partial f}{\partial \theta}(\theta_i)$$

Descente de gradient stochastique

Descente de gradient: $f(\theta) = -\sum_{i \in \text{toutes les données}} \log A_\theta(x_i)_{y_i}$
$\Rightarrow$ Problématique si beaucoup de données.

Descente de gradient stochastique

Descente de gradient: $f(\theta) = -\sum_{i \in \text{toutes les données}} \log A_\theta(x_i)_{y_i}$
$\Rightarrow$ Problématique si beaucoup de données.

- Pour tout i
  - Tirer (x, y) parmi les données disponibles
  - $f_i(\theta) = -\log A_\theta(x)_y$
  - $\theta_i = \theta_{i-1} - \mu \frac{\partial f}{\partial \theta}(\theta_{i-1})$

Descente de gradient stochastique
Version mini-batch

Version 1:

- Pour tout i
- Tirer K données (x_k, y_k) parmi les données disponibles
- $f_i(\theta) = -\frac{1}{K} \sum_k \log A_\theta(x_k)_{y_k}$
- $\theta_i = \theta_{i-1} - \mu \frac{\partial f}{\partial \theta}(\theta_{i-1})$

Descente de gradient stochastique
Version mini-batch

Version 1:
- Pour tout i
- Tirer K données (x_k, y_k) parmi les données disponibles
- $f_i(\theta) = -\frac{1}{K} \sum_k \log A_\theta(x_k)_{y_k}$
- $\theta_i = \theta_{i-1} - \mu \frac{\partial f}{\partial \theta}(\theta_{i-1})$

Version 2:
- $i = 1$
- Pour tout $j <$ nombre d'itérations sur l'ensemble des données,
  - Reinitialiser les données disponibles
  - Tant qu'il reste des données disponibles
    - Tirer K données (x_k, y_k) parmi les données disponibles et les retirer
    - $f_i(\theta) = -\frac{1}{K} \sum_k \log A_\theta(x_k)_{y_k}$
    - $\theta_i = \theta_{i-1} - \mu \frac{\partial f}{\partial \theta}(\theta_{i-1})$
  - $i+ = 1$

Le perceptron

Généralise la régression logistique.
Classifieur souple sur C classes.

Le perceptron

Généralise la régression logistique.
Classifieur souple sur C classes.

Hypothèse: $\log P(Y(X) = c|X = x) \propto S(\sum w_i^{(c)} x_i + b_i) = Q^{(c)}$
Avec S une fonction scalaire non linéaire, typiquement
$S(x) =$ sigmoïde$(x) = \frac{1}{1+\exp -x}$.

Le perceptron

Généralise la régression logistique.
Classifieur souple sur C classes.

Hypothèse: $\log P(Y(X) = c|X = x) \propto S(\sum w_i^{(c)} x_i + b_i) = Q^{(c)}$
Avec S une fonction scalaire non linéaire, typiquement
$S(x) =$ sigmoïde$(x) = \frac{1}{1+\exp -x}$.

Au final, $P(Y(X) = c|X = x) =$ softmax$((Q^{(k)})_k)(c)$

Le perceptron multicouches

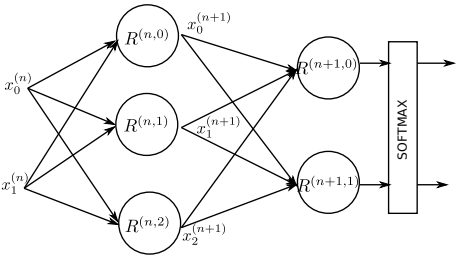
Classifieur souple sur C classes.

N couches

$x^{(0)}$=les données d'entrée.
$x^{(n)}$ =entrées de la n-ème couches,

Couche n M_n sorties
$R^{(n,m)}(x^{(n)}) = S(\sum w_k^{(n,m)} x_k^{(n)} + b_k^{(n,m)})$

Finish: fonction softmax



Optimisation par une implémentation particulière de la descente du gradient: l'algorithme de rétro-propagation

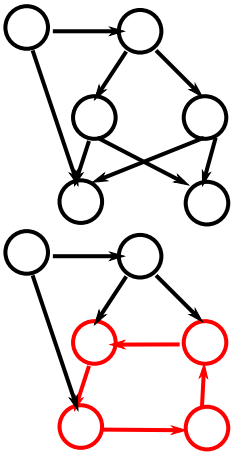
Réseaux de neurones et Graphes de calcul

Graphes

Un *graphe orienté* $G = (V, E)$ est une collection discrète d'objets V appelés *noeuds* et un ensemble de liens ou *arêtes* entre ces noeuds ($E \subset V \times V$).

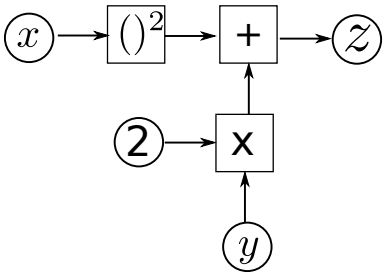
Un cycle de longueur n partant d est une suite finie d'arêtes $a_i = (x_i, y_i) \in E$, $y_i = x_{i+1}$ pour $i < n - 1$ et $y_{n-1} = x_0$.

Un *graphe orienté acyclique* (Directed Acyclic Graph) est un graphe orienté qui n'induit aucun cycle.



Graphe de calcul

Un graphe alterné est un graphe dont les noeuds peuvent appartenir à deux sous-ensembles V_0 et V_1 et dont chaque arête est dans $V_0 \times V_1$.
Un graphe de calcul est:
- un graphe
- orienté et acyclique
- dont les noeuds sont alternés entre *données* et *transformations*



Tenseurs

Les données d'un graphe de calcul sont des *tenseurs*:

Tenseurs

Les données d'un graphe de calcul sont des *tenseurs*:

Un tenseur T d'ordre N et de dimensions $[n_0, n_1, \cdots, n_{N-1}]$ sur un corps $\mathbb{K}(= \mathbb{R}$ ou $\mathbb{C})$ est défini par ses composantes $T_{i_0,i_1,\cdots,i_{N-1}} \in \mathbb{K}$, $0 \leq i_j < n_j$.
Ils forment un espace vectoriel de dimension $n_0 \times n_1, \times \cdots \times n_{N-1}$

Tenseurs

Les données d'un graphe de calcul sont des *tenseurs*:

Un tenseur T d'ordre N et de dimensions $[n_0, n_1, \cdots, n_{N-1}]$ sur un corps $\mathbb{K}(= \mathbb{R}$ ou $\mathbb{C})$ est défini par ses composantes $T_{i_0,i_1,\cdots,i_{N-1}} \in \mathbb{K}$, $0 \leq i_j < n_j$.
Ils forment un espace vectoriel de dimension $n_0 \times n_1, \times \cdots \times n_{N-1}$

- Les tenseurs d'ordre $N = 0$ correspondent aux scalaire (par convention)
- Les tenseurs d'ordre $N = 1$ correspondent aux vecteurs et aux signaux
- Les tenseurs d'ordre $N = 2$ correspondent aux matrices, aux "petits tableaux" et aux images

Si on a B tenseurs de dimensions identiques $[n_0, n_1, \cdots, n_{N-1}]$, on peut les regrouper dans un tenseur *batch* de dimensions

$$[B, n_0, n_1, \cdots, n_{N-1}]$$

Images: $[B, W, H]$, Signaux: $[B, T]$

Couramment une image $W \times H$ est représentée par un tenseur d'ordre 3, de dimensions $[C, W, H]$ avec C le nombre de *canaux* (exemple: canaux Red / Green / Blue).
De même un signal de durée T (1D) est représenté par un tenseur d'ordre 2, de dimensions $[C, T]$ avec C le nombre de *canaux* (exemple: son stéréo).

Images: $[B, C, W, H]$, Signaux: $[B, C, T]$

Opérations sur les tenseurs

On définit les opérations élémentaires op $+, /, \times$ entre deux tenseurs de même dimensions par

$$(S \operatorname{op} T)i_0, i_1, \cdots, i_{N-1} = S_{i_0,i_1,\cdots,i_{N-1}} \operatorname{op} T_{i_0,i_1,\cdots,i_{N-1}}$$

Opérations sur les tenseurs

On définit les opérations élémentaires op $+, /, \times$ entre deux tenseurs de même dimensions par

$$(S \operatorname{op} T)i_0, i_1, \cdots, i_{N-1} = S_{i_0,i_1,\cdots,i_{N-1}} \operatorname{op} T_{i_0,i_1,\cdots,i_{N-1}}$$

De même si f est une fonction de $\mathbb{K} \to \mathbb{K}$, on prolonge f aux tenseurs par:

$$(f(T))_{i_0,i_1,\cdots,i_{N-1}} = f(T_{i_0,i_1,\cdots,i_{N-1}})$$

Opérations sur les tenseurs

On définit les opérations élémentaires op $+, /, \times$ entre deux tenseurs de même dimensions par

$$(S \operatorname{op} T)i_0, i_1, \cdots, i_{N-1} = S_{i_0,i_1,\cdots,i_{N-1}} \operatorname{op} T_{i_0,i_1,\cdots,i_{N-1}}$$

De même si f est une fonction de $\mathbb{K} \to \mathbb{K}$, on prolonge f aux tenseurs par:

$$(f(T))_{i_0,i_1,\cdots,i_{N-1}} = f(T_{i_0,i_1,\cdots,i_{N-1}})$$

Les opérations linéaires de l'espace des tenseurs de dimensions données vers un autre espace de tenseurs est un espace vectoriel de dimension le produit de celles des espaces considérés.

Opérations sur les tenseurs

Concaténation:
Si T_0 de dimensions $[n_0, n_1, \cdots n_{N-2}, n_{0,N-1}]$ et
T_1 de dimensions $[n_0, n_1, \cdots n_{N-2}, n_{1,N-1}]$ alors on définit
la concaténation de T_0 et T_1 le long de la dimension $N-1$ par
$\text{cat}(T_0, T_1|N-1)$ est le tenseur de dimensions $[n_0, n_1, \cdots n_{N-2}, n_{0,N-1} + n_{1,N-1}]$
tel que

$$\text{cat}(T_0, T_1|N-1)_{i_0,\cdots,i_{N-1}} = (T_0)_{i_0,\cdots,i_{N-1}} \quad \text{si} \quad i_{N-1} < n_{0,N-1}$$

$$\text{cat}(T_0, T_1|N-1)_{i_0,\cdots,i_{N-1}} = (T_1)_{i_0,\cdots,i_{N-1}-n_{0,N-1}} \quad \text{sinon}$$

Opérations sur les tenseurs

Réduction:
Une opération de réduction selon la dimension $N-1$ est une fonction qui à un tenseur T de dimensions $[n_0, n_1, \cdots, n_{N-1}]$ associe un tenseur d'ordre $N-1$ et de dimensions $[n_0, n_1, \cdots, n_{N-2}]$

Opérations sur les tenseurs

Réduction:
Une opération de réduction selon la dimension $N-1$ est une fonction qui à un tenseur T de dimensions $[n_0, n_1, \cdots, n_{N-1}]$ associe un tenseur d'ordre $N-1$ et de dimensions $[n_0, n_1, \cdots, n_{N-2}]$

Par exemple:

$$\text{somme}(T|N-1)_{i_0,\cdots,i_{N-2}} = \sum_{0 \leq j < n_{N-1}} T_{i_0,\cdots,i_{N-2},j}$$

Apprentissage de Réseaux de neurones

Réseau de neurones

Un graphe de calcul paramétrique est un graphe de calculs dont tout ou partie des transformations dépend d'un ensemble de paramètres θ.

Réseau de neurones

Un graphe de calcul paramétrique est un graphe de calculs dont tout ou partie des transformations dépend d'un ensemble de paramètres θ.

Un réseau de neurones est un graphe de calcul paramétrique dont tout ou partie des transformations est non-linéaires

Algorithme de rétropropagation du gradient

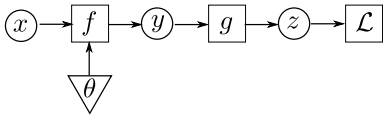
Pour régler les paramètres libres θ d'un réseau de neurones, on utilise une variation de la descente du gradient tirant partie de la structure de graphe de calcul pour calculer efficacement le gradient.

Nota: ici gradient = application linéaire tangente

Algorithme de rétropropagation du gradient

Cas 1: Gradient par rapport à θ de

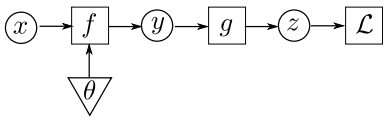
$$l(x, \theta) = \mathcal{L}(h(g(f(x|\theta))))$$



Algorithme de rétropropagation du gradient

Cas 1: Gradient par rapport à θ de

$$l(x, \theta) = \mathcal{L}(h(g(f(x|\theta))))$$

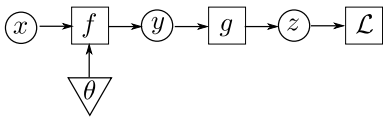


$$f(x|\theta + \delta\theta) = f(x|\theta) + \frac{\partial f}{\partial \theta}(x, \theta)\delta\theta + o(\delta\theta)$$
$$g(f(x|\theta + \delta\theta)) = g(f(x|\theta)) + \frac{\partial g}{\partial X}(f(x, \theta)) \circ \frac{\partial f}{\partial \theta}(x, \theta)\delta\theta + o(\delta\theta)$$

Algorithme de rétropropagation du gradient

Cas 1: Gradient par rapport à θ de

$$l(x, \theta) = \mathcal{L}(h(g(f(x|\theta))))$$



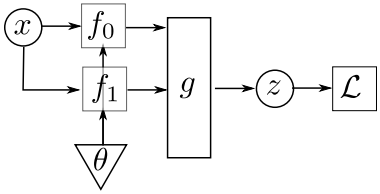
$$f(x|\theta + \delta\theta) = f(x|\theta) + \frac{\partial f}{\partial \theta}(x, \theta)\delta\theta + o(\delta\theta)$$
$$g(f(x|\theta + \delta\theta)) = g(f(x|\theta)) + \frac{\partial g}{\partial X}(f(x, \theta)) \circ \frac{\partial f}{\partial \theta}(x, \theta)\delta\theta + o(\delta\theta)$$

$$l(x, \theta + \delta\theta) = l(x, \theta) + \underbrace{\frac{d\mathcal{L}}{dX}(g(f(x|\theta))) \circ \frac{\partial g}{\partial X}(f(x, \theta)) \circ \frac{\partial f}{\partial \theta}(x, \theta)}_{\frac{\partial l}{\partial \theta}}\delta\theta + o(\delta\theta)$$

Algorithme de rétropropagation du gradient

Cas 2: Gradient par rapport à θ de

$$l(x, \theta) = \mathcal{L}(g(f_0(x|\theta), f_1(x|\theta)))$$

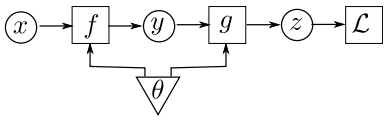


$$l(x, \theta + \delta\theta) =$$
$$l(x, \theta) + \sum_{i=0}^{1} \frac{d\mathcal{L}}{dX} g(f_0(x|\theta), f_1(x|\theta)) \circ \frac{\partial g}{\partial X_i}(f_0(x, \theta), f_1(x, \theta)) \circ \frac{\partial f_i}{\partial \theta}(x, \theta)\delta\theta + o(\delta\theta)$$

Algorithme de rétropropagation du gradient

Cas 3: Gradient par rapport à θ de

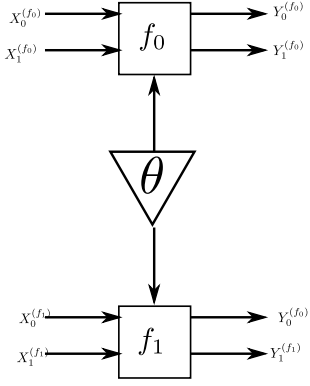
$$l(x, \theta) = \mathcal{L}(g(f(x|\theta)|\theta))$$

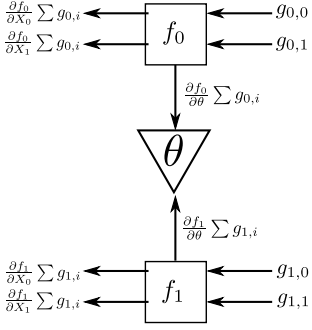


$$l(x, \theta + \delta\theta) =$$
$$l(x, \theta) + \frac{d\mathcal{L}}{dX} g(f(x|\theta)|\theta) \circ \left(\frac{\partial g}{\partial X}(f(x|\theta)|\theta) \circ \frac{\partial f}{\partial \theta}(x, \theta) + \frac{\partial g}{\partial \theta}(f(x|\theta)|\theta) \right) \delta\theta + o(\delta\theta)$$

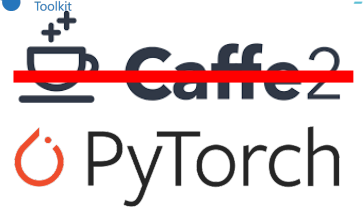
Algorithme de rétropropagation du gradient

On suppose que pour toute fonction $f(X_0, X_1, \cdots, X_{N-1})$ apparaissant dans le graphe de calcul on sait calculer $\frac{\partial f}{\partial X_i}$, $\forall i$. (X_i donnée ou paramètre).





Frameworks de calcul



- Calcul tensoriel
- Graphes de calcul
- Différentiation automatique
- Accélération matérielle

Exemples PyTorch
Opération sur les tenseurs

```
#Création d'un tenseur
tenseur = torch.zeros((10,5,100)) # [10,5,100]
tenseur = torch.ones((10,5,100)) # [10,5,100]
tenseur = torch.rand((10,5,100)) # [10,5,100]

tenseur_zeros = torch.zeros_like(tenseur) # [10,5,100]
tenseur_copy = torch.clone(tenseur)

print(f'Tenseur de dimensions {tenseur.shape} et de type {tenseur.dtype}')

# Opération élémentaires
tenseur2 = torch.rand((10,5,100))  # [10,5,100]
tenseur_au_carre = tenseur**2  # [10,5,100]
tenseur_exp = torch.exp(tenseur)  # [10,5,100]
tenseur_somme = tenseur + tenseur2  # [10,5,100]
tenseur_produit = tenseur * tenseur2  # [10,5,100]

# Concaténation
tenseur_concatene_axe0 = torch.cat([tenseur, tenseur2],axis=0)  # [20,5,100]
tenseur_concatene_axe1 = torch.cat([tenseur, tenseur2],axis=1)  # [10,10,100]
tenseur_concatene_axe2 = torch.cat([tenseur, tenseur2],axis=2)  # [10,5,200]

tenseur_tile = torch.tile(tenseur,dims=(2,3,4))  # [20,15,400]
```

Exemples PyTorch
Opération sur les tenseurs

```
# Opérations de réduction
tenseur_reduit_axe0 = torch.mean(tenseur,axis=0)  # [5,100]
tenseur_reduit_axe1 = torch.mean(tenseur,axis=1)  # [10,100]
tenseur_reduit_axe2 = torch.mean(tenseur,axis=2)  # [10,5]

tenseur_reduit_axe0_kd = torch.mean(tenseur,axis=0,keepdim=True)  # [1,5,100]

#ãManipulation sur les dimensions

tenseur_21 = torch.transpose(tenseur,1,2) # [10,100,5]
tenseur_flat = torch.flatten(tenseur) # [5000,]
tenseur_newdim_0 = torch.unsqueeze(tenseur,dim=0) # [1,10,5,100]
tenseur_newdim_1 = torch.unsqueeze(tenseur,dim=1) # [10,1,5,100]

# Slicing
tenseur_partie = tenseur[0,:,:] # [5,100]
tenseur_partie = tenseur[0,...] # [5,100]
tenseur_partie = tenseur[0:5,:,:] # [5,5,100]
tenseur_partie = tenseur[0,:,0::2]# [10,5,50]
```

Exemples PyTorch
Optimisation d'un graphe de calcul - régression linéaire

```
a_true = 2.0
b_true = -1.0

x = torch.rand((10,)) # simulation des données
y = a_true * x + b_true

a = torch.zeros(1,requires_grad=True) # les objets que l'on va faire converger vers les valeurs recherchées
b = torch.zeros(1,requires_grad=True) # requires_grad => un champs gradient est attaché à l'objet crée

for i in range(1000):

    y_est = a*x + b # je connais x, j'estime y / paramètres estimés courants
                    # cette ligne crée un graphe de calcul entre x et y_est mettant en jeu a et b

    loss = torch.mean((y_est-y)**2) # je calcule l'erreur entre l'estimation et les valeurs observées

    loss.backward() # je différencie la fonction de perte
                    # cela entraîne la différentiation automatique de tout le graphe de calcul
                    # le gradient est mis à jour dans toutes les variables du graphe / requires_grad = True
    print(f'Itération {i}:')
    print(f"a: {a.item():.4f}, b: {b.item():.4F}, loss: {loss.item():.4f},  grad a: {a.grad.item():.4f}, grad b: {b.grad.item():.4f}")

    with torch.no_grad(): #je vais effectuer des opérations sur des objets attachés au graphe de calcul
                          # mais je ne veux pas que ces opérations entrent dans l'optimisation des paramètres
        a -=  0.1*a.grad # descente de gradient de pas 0.1
        b -=  0.1*b.grad

        a.grad.zero_() # je remets à 0 tous les champs gradient des objets
        b.grad.zero_()
```

Modèles et couches de réseaux de neurones - niveau 1

Réprésentation objet d'un réseau de neurones

Dans les frameworks de calcul, un réseau de neurones =*Modèle* défini par:

- un ensemble de paramètres
- une fonction *forward* des tenseurs d'entrée $\mapsto$ des tenseurs de sortie.
    - la fonction forward alterne des opération linéaires et des opérations non linéaires

Dans certains cas simples (mais nombreux), le modèle = succession de modèles plus simples *couches (layers)*

Exemples PyTorch
Définition d'un réseau de neurones

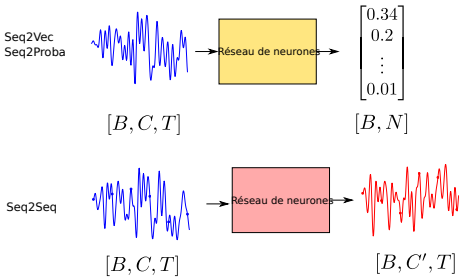
```
class MonModeleQuiTorche(torch.nn.Module):
def __init__(self,delta_chan=4):

    torch.nn.Module.__init__(self)

    self.delta_chan=delta_chan
    self.learnable_param = torch.nn.Parameter(torch.rand([1,delta_chan,1]))
    self.not_learnable_param = torch.rand((1,delta_chan,1))

def forward(self,x): #x is [B,input_chan,T]
    # output is [B,self.output_chan,T]

    x_reduced = torch.mean(x , axis = 1 , keepdim=True)
    x_duplicated = torch.tile(x_reduced , dims = (1, self.delta_chan, 1))

    y0 = self.learnable_param *x_duplicated
    y1 = y0+ self.not_learnable_param
    y2 = torch.abs(y1)
    y3 = torch.concat([x, y2], axis=1)

    return y3

def __call__(self,x):
# Défini dans la classe mère
    return self.forward(x)

mon_modele=MonModeleQuiTorche(delta_chan=4)
x= torch.rand(5,1,100)
y = mon_modele(x)
z = mon_modele(y)
```

Exemples PyTorch
Définition d'un réseau de neurones

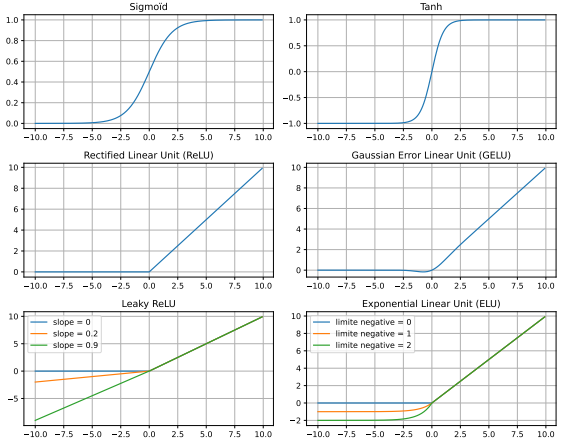
```
mon_modele_sequentiel = torch.nn.Sequential(
  MonModeleQuiTorche(4),
  MonModeleQuiTorche(5),
  MonModeleQuiTorche(6)  )
print(mon_modele_sequentiel(torch.rand(5,1,100)).shape)

#Quand il y a beaucoup de couches

mlp = torch.nn.Sequential(*[MonModeleQuiTorche(4+i) for i in range(4)])
mlp(torch.rand(10))
```

Réseaux de neurones et signal

Réseau de neurones = grosse fonction.
Signal = tenseur [B , C, T]



Fonctions non linéaires



$\Rightarrow$ Quand ils existent les paramètres des non linéarités peuvent être appris.

Couches linéaires = couches denses (=perceptron des familles)

A un tenseur X de dimensions $[B, C_{\text{in}}]$ associe Y de dimensions $[B, C_{\text{out}}]$.

Paramètres:
- les poids : tenseur d'ordre 2, W, $[C_{\text{in}}, C_{\text{out}}]$
- les biais: tenseur d'ordre 1, μ, $[C_{\text{out}}]$

$$\forall 0 \leq b < B,\ 0 \leq j < C_{\text{out}}, \quad Y_{b,j} = \sum_i W_{i,j} X_{b,i} + \mu_j$$

Nombre de paramètres: $C_{\text{in}} \times C_{\text{out}} (+C_{\text{out}})$

```
linear = torch.nn.Linear(in_features=4,
                         out_features=10
                         )
```

Couches linéaires appliquées à un signal

Approche pour appliquer une couche linéaire à un tenseur $[B, C_{\text{in}}, T]$:

$$[B, C_{\text{in}}, T] \rightarrow \text{flatten} \rightarrow [B, C_{\text{in}} \times T] \rightarrow [B, C_{\text{out}}]$$

Nombre de paramètres: $T \times C_{\text{in}} \times C_{\text{out}} (+C_{\text{out}})$

Couches de convolutions

Existe en trois parfums: Conv1D , Conv2D, Conv3D.

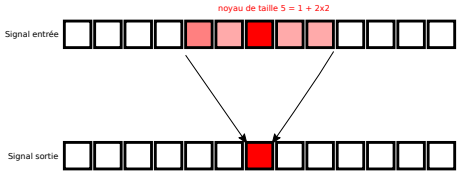
La Conv1D associe à un tenseur X de dimensions $[B, C_{\text{in}}, T_{\text{in}}]$ un tenseur Y de dimension $[B, C_{\text{out}}, T_{\text{out}}]$.
$\Rightarrow$ Fait "glisser" une couche dense le long de l'axe temporel.

Paramètre: noyaux $[C_{\text{out}}, C_{\text{in}}, K]$, $K = 2k + 1 =$ taille du noyau

```
conv = torch.nn.Conv1d(in_channels=1,  # entrée [B,1,T]
                       out_channels=4, # sortie [B,4,T]
                       kernel_size=11, # préférer les nombres impairs
                       padding='same', # même effet que (kernel_size-1)//2
                       )
```

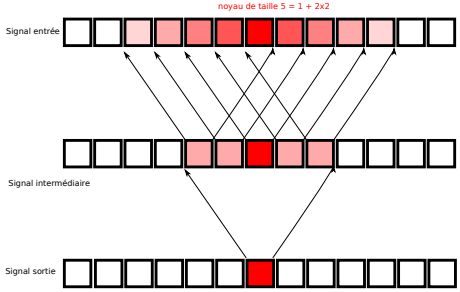
Intermzzo : champs receptif

Intervalle d'échantillons dans le signal d'entrée qui ont participé au calcul d'un échantillon dans le signal de sortie.
ă Pour une convolution de noyau 5:

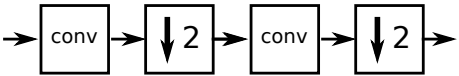


Intermzzo : champs receptif

Intervalle d'échantillons dans le signal d'entrée qui ont participé au calcul d'un échantillon dans le signal de sortie.
ă Pour 2 convolutions de noyau 5:



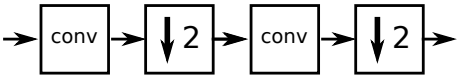
Couches de convolutions



Convolution + décimation $\leftrightarrow$ Convolution avec un *stride* de 2

```
conv = torch.nn.Conv1d(in_channels=1,  # entrée [B,1,T]
                       out_channels=4, # sortie [B,4,T']
                       kernel_size=11, # préférer les nombres impairs
                       stride=2,       # T' = T//2
                       padding='same', # idem (kernel_size-1)//2
                       )
```

Couches de convolutions



Convolution + décimation $\leftrightarrow$ Convolution avec un *stride* de 2

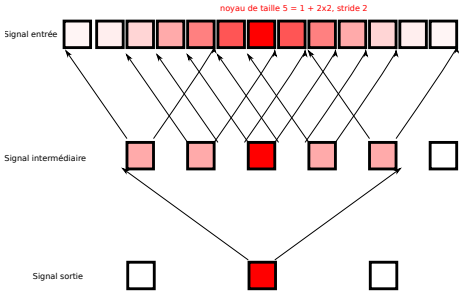
```
conv = torch.nn.Conv1d(in_channels=1,  # entrée [B,1,T]
                       out_channels=4, # sortie [B,4,T']
                       kernel_size=11, # préférer les nombres impairs
                       stride=2,       # T' = T//2
                       padding='same', # idem (kernel_size-1)//2
                       )
```

Nombre de paramères: taille noyau $\times$ $C_{\text{in}} \times C_{\text{out}} (+C_{\text{out}})$

Intermzzo : champs receptif

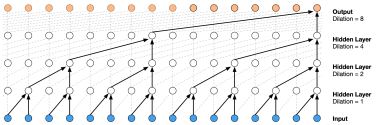
Intervalle d'échantillons dans le signal d'entrée qui ont participé au calcul d'un échantillon dans le signal de sortie.

ă Pour 2 convolutions de noyau 5:



Convolution dilatées

D'une couche à une autre, "dilate" le noyau des convolution (ie le bourre de 0)
Solution pour agrandir le champs perceptif sans décimer le signal




```
pointwise = torch.nn.Conv1d(in_channels=1,   # entrée [B,1,T]
                            out_channels=4,  # sortie [B,4,T]
                            kernel_size=11,   # préférer les nombres impairs
                            dilation=2, # paramètre de dilatation
                            padding='same'
                           )
```

Pointwise convolutions

Convolution de noyau de taille 1
$\Rightarrow$ Couche dense appliquée indépendamment échantillon par échantillon

```
pointwise = torch.nn.Conv1d(in_channels=1,   # entrée [B,1,T]
                            out_channels=4,  # sortie [B,4,T]
                            kernel_size=1,   # préférer les nombres impairs
                            stride=1,
                           )
```

Depthwise convolutions

Une convolution indépendante sur chacun des canaux du signal d'entrée.

```
depthwise = torch.nn.Conv1d(in_channels=4,  # entrée [B,1,T]
                            out_channels=4,  # sortie [B,4,T]
                            groups= 4,       # correspond à in_channels
                            kernel_size=11,  # préférer les nombres impairs
                            stride=1,
                            )
```

Convolution séparables

Cascade d'une convolution "pointwise" et d'une convolution "depthwise".

```
separable_convolution = torch.nn.Sequential(depthwise,
                                            pointwise)
```

Nombre de paramètres: taille noyau $\times C_{\text{in}} + C_{\text{in}} \times C_{\text{out}}(+2 \times C_{\text{out}})$

Réseaux de neurones récurrents

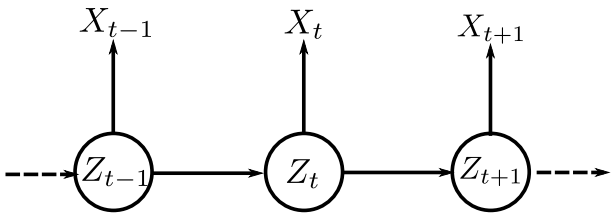
Système dynamique :
- x = entrée
- y = sortie ou observable
- h = état interne ou état caché

$$h(t+dt) = F(x(t), h(t))dt$$
$$y(t) = G(h(t))$$

cf Chaînes de MArkov cachées

Modèles de Markov cachés

Un *modèle de Markov caché* est une chaîne de Markov à laquelle on ajoute pour chaque temps t une variable dite de diffusion X_t.
La loi de X_t est définie conditionnellement à Z_t: $P(X_t|Z_t)$.

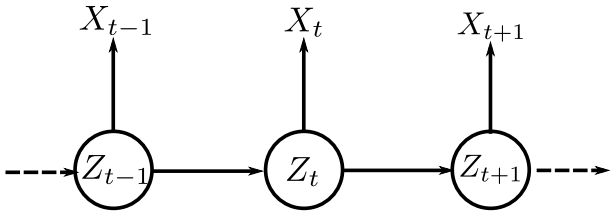


Modèles de Markov cachés - Propriétés

On a encore une propriété d'indépendance conditionnelle:

$$P(X_{t-k}, X_{t+k}|Z_t) = P(X_{t-k}|Z_t)P(X_{t+l}|Z_t)\,\forall t, k, l > 0, k \leq t$$

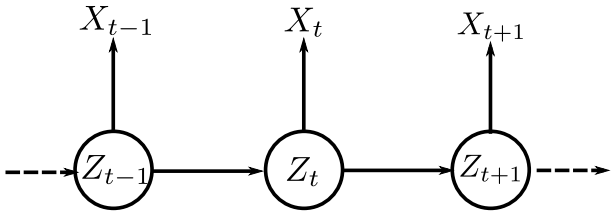
Modèles de Markov cachés - Problèmes d'inférence



On a deux problèmes d'inférence sur les modèles de Markov cachés:

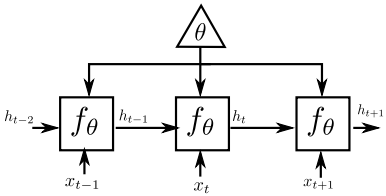
- Estimer $P(Z_t = z|x_{0:T-1}), \forall z \in \mathcal{E}, \forall t < T$

Modèles de Markov cachés - Problèmes d'inférence



On a deux problèmes d'inférence sur les modèles de Markov cachés:

- Estimer $P(Z_t = z|x_{0:T-1}), \forall z \in \mathcal{E}, \forall t < T$
- Estimer la séquence $(z^*_{0:T-1}) = \text{argmax}(P(z_{0:T-1}|x_{0:T-1})$



- Définissent un état caché h_i pour chaque échantillon x_i
- $h_i = f(h_{i-1}, x_i, \theta)$
- Partage de θ à travers le temps

RNN *vanilla*: $\theta = (W, b)$

$$f_\theta(x, h) = \text{NL}(W \cdot \text{cat}(x, h) + b)$$

avec typiquement $\text{NL} = \tanh$

Réseaux de neurones récurrents
Principe de base

```
recurrent = torch.nn.RNN(input_size=6, # x is [B, T , input_size] ! Temps et canaux inversés
                         hidden_size =15, # h is [B,T, hidden_size]
                         batch_first=True,  # pour que la première dimension soit bien le batch
                         num_layers =1, # Paramètre par défaut
                         bidirectional= False # Paramètre par défaut
                         )
x = torch.rand([10, 100 , 6])
h , h_layers_end = recurrent(x)
print(h.shape) #[10 , 100 , 15]
print(h_layers_end.shape)#[1,15,10]
```

```
recurrent = torch.nn.RNN(input_size=6, # x is [B, T , input_size] ! Temps et canaux inversés
                         hidden_size =15, # h is [B,T, hidden_size]
                         batch_first=True,  # pour que la première dimension soit bien le batch
                         num_layers =4, # Le h de la couche n devient le x de la couche n+1
                         bidirectional= False # Paramètre par défaut
                         )
x = torch.rand([10, 100 , 6])
h , h_layers_end = recurrent(x)
print(h.shape) #[10 , 100 , 15] # la séquence des états cachés de la dernière couche
print(h_layers_end.shape)#[4,15,10] # le dernier état caché de chaque couche
```

Deux modes d'utilisation :

- Seq2Seq : $x \rightarrow h$: on garde la séquence des états cachés
- Intégrateur : $x \rightarrow h[:, -1, :]$: on garde le dernier état caché

Réseau bidirectionnel:
- On applique le réseau à la séquence x
- On applique un réseau identique à la même séquence mais renversée dans le temps : $\tilde{x} = (x_{N-1}, \cdots x_0)$
- Remarque: les poids des réseaux directs et renversés sont différents
- On concatène les résultats

```
recurrent = torch.nn.RNN(input_size=6, # x is [B, T , input_size] ! Temps et canaux inversés
                         hidden_size =15, # h is [B,T, hidden_size]
                         batch_first=True,  # pour que la première dimension soit bien le batch
                         num_layers =4, # Le h de la couche n devient le x de la couche n+1
                         bidirectional= True # Paramètre par défaut
                         )
x = torch.rand([10, 100 , 6])
h , h_layers_end = recurrent(x)
print(h.shape) #[10 , 100 , 15*2] # les séquence des états cachés de la dernière couche en sens direct et inversé concaténés
print(h_layers_end.shape)#[4,15,10] # le dernier état caché de chaque couche
```

Un mode d'utilisation :
- Seq2Seq : $x \to h$: on garde la séquence des états cachés

Chaînes de Markov cachées - Algorithme Forward Backward

Estimer $P(Z_t = z | x_{0:T-1}), \forall z \in \mathcal{E}$

$$\gamma_t(z) = P(Z_t = z | X_{0:T-1} = x_{0:T-1}) = \frac{P(Z_t = z \cap X_{0:T-1} = x_{0:T-1})}{P(X_{0:T-1} = x_{0:T-1})}$$

$$\gamma_t(z) = \frac{1}{\Omega} P(Z_t = z \cap X_{0:t} = x_{0:t}) \cdot P(X_{t+1:T-1} = x_{t+1:T-1} | Z_t = z \cap X_{0:t} = x_{0:t})$$

$$\gamma_t(z) = \frac{1}{\Omega} \underbrace{P(Z_t = z \cap X_{0:t} = x_{0:t})}_{f_{0:t}(z)} \cdot \underbrace{P(X_{t+1:T-1} = x_{t+1:T-1} | Z_t = z)}_{b_{t:T-1}(z)}$$

- $f_{0:t}(z)$ se calcule par récurrence:

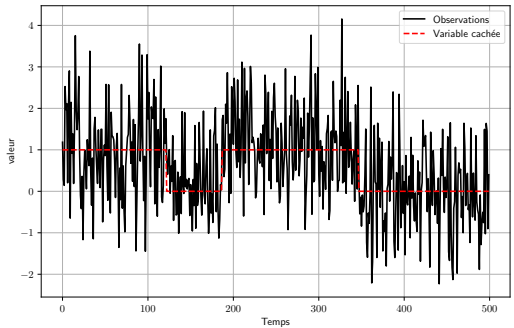
$\forall z,\ f_{0:t}(z) = \sum_y f_{0:t-1}(y) \cdot \Pi(y \to z) \cdot P(X_t = x_t | Z_t = z)$

- $b_{t:T-1}(z)$ se calcule par récurrence aussi:

$\forall z,\ b_{t:T-1}(z) = \sum_y b_{t+1:T-1}(y) \cdot \Pi(z \to y) \cdot P(X_{t+1} = x_{t+1} | Z_{t+1} = y)$

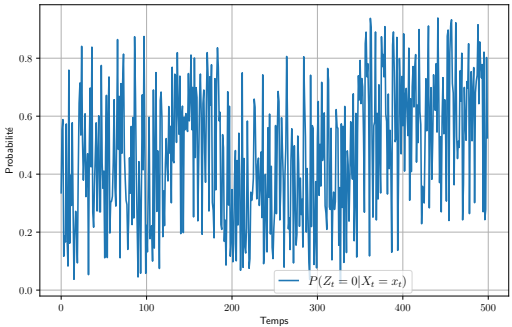
Chaînes de Markov cachées - Algorithme Forward Backward

Deux états, A et B, $\Pi(A \to A) = \Pi(B \to B) = 0.99$, $(Y_t|Z_t = A) \sim \mathcal{N}(0,1)$, $(Y_t|Z_t = B) \sim \mathcal{N}(1,1)$



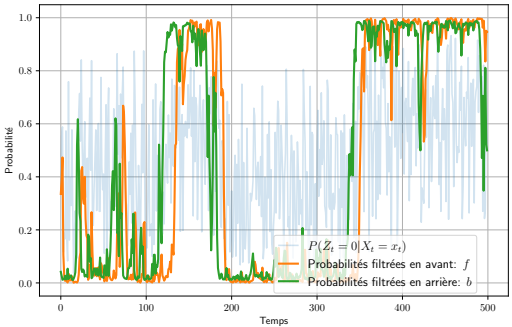
Chaînes de Markov cachées - Algorithme Forward Backward

Deux états, A et B, $\Pi(A \to A) = \Pi(B \to B) = 0.99$, $(Y_t|Z_t = A) \sim \mathcal{N}(0,1)$, $(Y_t|Z_t = B) \sim \mathcal{N}(1,1)$



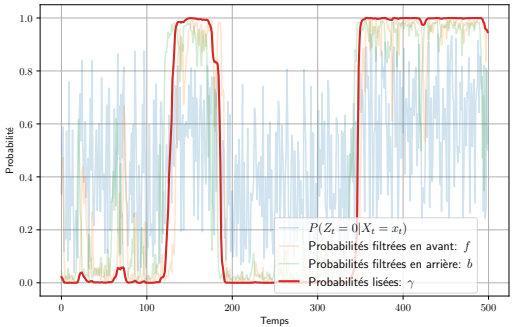
Chaînes de Markov cachées - Algorithme Forward Backward

Deux états, A et B, $\Pi(A \to A) = \Pi(B \to B) = 0.99$, $(Y_t|Z_t = A) \sim \mathcal{N}(0, 1)$, $(Y_t|Z_t = B) \sim \mathcal{N}(1, 1)$

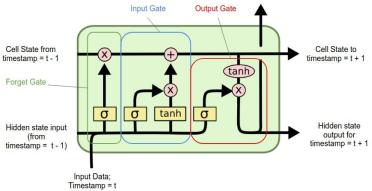


Chaînes de Markov cachées - Algorithme Forward Backward

Deux états, A et B, $\Pi(A \to A) = \Pi(B \to B) = 0.99$, $(Y_t|Z_t = A) \sim \mathcal{N}(0, 1)$, $(Y_t|Z_t = B) \sim \mathcal{N}(1, 1)$

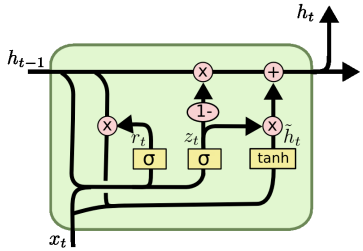


Structure élaborée de réseau récurrent pour favoriser la "circulation" de l'information à travers le temps




```
lstm = torch.nn.LSTM(input_size=4, # x is [B, T , input_size]
                     hidden_size =10, # h is [B,T, hidden_size]
                     num_layers =2, # h de la couche 0 devient le x de la couche 1 etc.
                     batch_first=True,  # pour que la première dimension soit bien le batch
                     bidirectional= True # cf Algorithme forward backward
                     )
```

Simplification du LSTM, limite le nombre de paramètres libres




```
gru = torch.nn.GRU(input_size=4, # x is [B, T , input_size]
                   hidden_size =10, # h is [B,T, hidden_size]
                   num_layers =2, # h de la couche 0 devient le x de la couche 1 etc.
                   batch_first=True,  # pour que la première dimension soit bien le batch
                   bidirectional= True # cf Algorithme forward backward
                   )
```

Couches d'Attention

Popularisé en NLP par Vaswani et Al., *Attention is all you need*,NeurIPS 2017 .

Idée de base: transformer (x_i) en (y_i) selon:

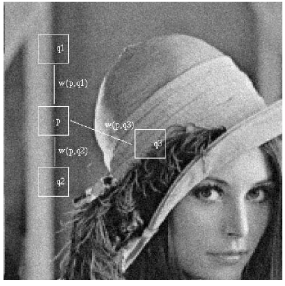
$$y_i = \sum_i \langle x_i, x_j \rangle x_j$$

On ajoute des paramètres pour gagner en expressivité
$q_i = Q(x_i) \quad k_i = K(x_i), \quad v_i = V(x_i)$ avec Q, K, V à apprendre.
Puis on pose $P_i = \text{softmax}((\langle q_i, k_j \rangle)_j)$ et finalement

$$y_i = \sum_i P_{ij} v_j$$

Couches d'Attention

A la base des Transformers et des LLM actuels.
Mais similarités avec l'idée des *Non Local Means* en débruitage d'images



Buades A., Morel J.M., *A non-local algorithm for image denoising*,CVPR 2005 .

Réseaux de neurones avec injection de connaissance

⇒ On peut injecter de la connaissance "métier" dans la structure d'un réseau de neurones:

Exemple: sinc-net pour la reconnaissance de locuteur Ravanelli et Al., *Speaker Recognition from raw waveform with SincNet*,2018

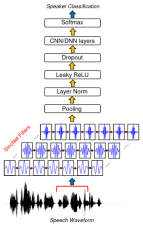
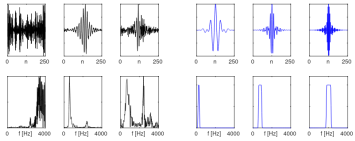


Fig. 1: Architecture of SincNet.

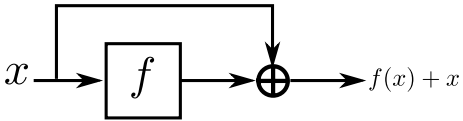


Exemple: deep-unfolding principe général qui consiste à débrayer certains paramètres dans un algorithme "classique" et de les apprendre.

Hershey, Le Roux et Al., *Deep Unfolding: Model-Based Inspiration of Novel Deep Architectures*, 2014

Trucs et astuces

Connexions résiduelles



Ajoute un "court circuit" entre les entrées et la sortie d'une couche ou d'une séquence de couches

- Lors de l'apprentissage, favorise la remontée du gradient
- Chaque couche apporte une "petite modification" des données
- Simplifie l'initialisation des paramètres de f: autour de 0

Batch normalization

Couche de normalisation: chargée de faire en sorte qu'à l'inférence, les données en entrée d'une couche soient statistiquement centrées et réduites.

Problème: ces statistiques varient avec les paramètres du réseau et elles doivent être *apprises* conjointement.

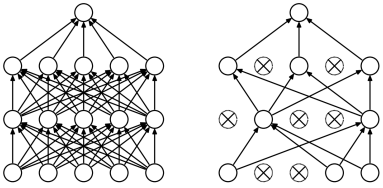
Une couche de *batch normalization*:
- Lors de l'entrainement: estime au fil des itérations les statistiques des données au moyen de moyennes mobiles
- Lors de l'inférence: normalise les données avec les statistiques apprises

```
batchnorm = torch.nn.BatchNorm1d(num_features=4 # En entrée [B,4,T]
                                 momentum=0.1 # intertie dans le calcul
                                              #de la moyenne mobile lors de l'apprentissage
                                 )
```

Dropout

- A l'entrainement supprimer une partie aléatoire des données en entrée ou des paramètres d'une couche
- Désactivé après l'entrainement
- Solution au surapprentissage

$\Rightarrow$ Surtout utilisée avant les couches très concentrées en paramètres: récurrentes, linéaires.



(a) Standard Neural Net (b) After applying dropout.

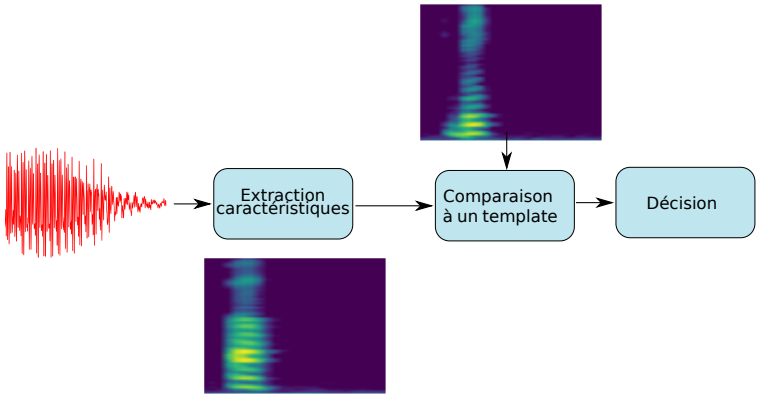
```
dropout = torch.nn.Dropout(p=0.2) # met 20 prct des composantes du tenseur d'entrée à 0
gru = torch.nn.GRU(input_size=4, # x is [B, T , input_size]
                   hidden_size =10, # h is [B,T, hidden_size]
                   dropout = 0.2 # dropout sur tous les poids
                   )
```

Application : reconnaissance vocale

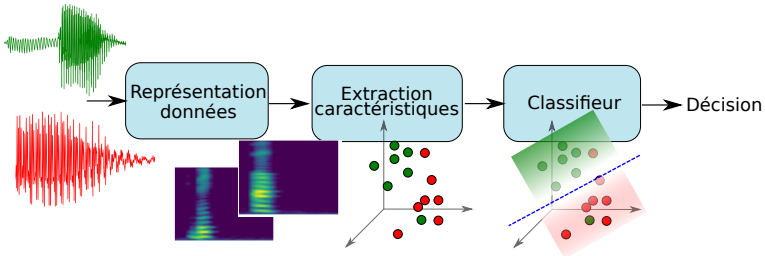
Tâches de classification en audio

- Détection de parole
- Reconnaissance de phonèmes
- Reconnaissance de mots-clef
- Reconnaissance de langue
- Reconnaissance d'émotion
- Détection de stress
- Reconnaissance d'environnement / ambiance / bruit de fond
- Reconnaissance de locuteur

Reconnaissance vocale - approches classiques



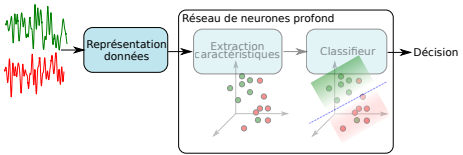
Reconnaissance vocale - approches par apprentissage

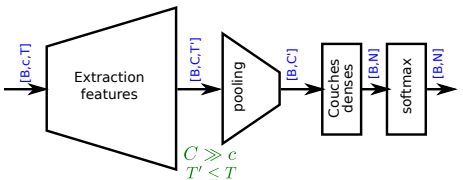


Classification de signaux

Trois étapes:
- Extraction de features temporelles
- Contraction dimension temporelle
  - Par moyenne
  - Statistiques d'ordres suppérieurs
  - Minimum, maximum etc.
  - Moyenne pondérée par attention
- Classification à proprement parler

Fonction de perte:
negative log vraisemblance
(torch.nn.NLLLoss())





Mise en oeuvre avec PyTorch

Gestion des données

```
class MyDataset(torch.utils.data.Dataset):

  def __init__(self, path_to_data):
      ...
  def __len__(self): #returns int
      ...
  def __getitem__(self,i): #returns (data_i,label_i)
      ...

dataset = MyDataset(...)

dataloader = DataLoader(dataset,
                  batch_size=10,
                  shuffle=True
                  )
```

Gestion des données

```
class MySampler(
  torch.utils.data.Sampler):

      def ___init___(self,...):
          ...
      def ___len___(self):
          #returns number of batches

      def ___getitem___(self,i):
          #returns list of indices in batch i

def my_collate_fn(list_of_x_y):
    #inputs : list of raw (data,label)
    # has to pack data into tensors
    # may perform data augmentation
    #returns (tensor of data , tensor of labels)

dataloader = DataLoader(dataset,
                        sampler=my_sampler,
                        collate_fn=my_collate_fn,
                        )
```

Entrainement

```
class metric_logger:

  def __init__(self,...):
      ...

  def reset_metrics(self):
      ...

  def update_metrics(self, batch_x,batch_y_true,batch_y_pred):
      ...

      return {'metric0':...,
              'metric1':...
              }

  def log(self):
      ...

device = 'cpu' # set so 'cuda:xx' if you have a GPU, xx is GPU index
model = ...
optimizer = torch.optim.Adam(model.parameters())
n_epochs=100

model.to(device)

metric_logger_train = metric_logger(...)
metric_logger_valid = metric_logger(...)
```

Entrainement

```
for epoch in range(n_epochs):
  metric_logger_train.reset()
  metric_logger_valid.reset()

  for batch_x,batch_y in dataloader_train:

      batch_x.to(device)
      batch_y.to(device)

      optimizer.zero_grad()

      batch_y_predicted = model(batch_x)

      l = loss(batch_y_predicted, batch_y)

      metric_logger_train.log(batch_x,batch_y,batch_y_predicted)

      l.backward()

      optimizer.step()

  for batch_x,batch_y in dataloader_valid:

      batch_x.to(device)
      batch_y.to(device)

      with torch.no_grad():
          batch_y_predicted = model(batch_x)

      metric_logger_valid.log(batch_x,batch_y,batch_y_predicted)
```